Columbia University, Fu Foundation School of Engineering and Applied Sciences
COMS 6698.013: Cloud Computing & Big Data Systems
Fall 2021

Mint Condition

Final Report

Backend Team

Adam Carpentieri | **ac4409**@columbia.edu

Bharathi Saravanabhavan | **bs3363**@columbia.edu

Gursifath Bhasin | **gb2760**@columbia.edu

Taku Takamatsu | **tt2828**@columbia.edu

Frontend Team

I Hun Chan | **ic2552**@columbia.edu

Jason William Don | **jwd2139**@columbia.edu

ML Team

Anirudh Birla | **ab5188**@columbia.edu

Charles Chen | **lc3533**@columbia.edu

Maxime Faucher | **mf3382**@columbia.edu

Pin-Chun Chen | **pc2939**@columbia.edu

Kushagra Jain | **kj2558**@columbia.edu

Project Repository: https://github.com/COMSE6998-013-mint-condition/mint-condition

INTRODUCTION

The market for sports memorabilia, especially the sale of trading cards, has been growing every year with an estimated market value of over $5 billion. This year alone, eBay has reported that more trading cards have been sold in the first six-months than in all of 2020; an estimated total of $2 billion in gross merchandise revenue. The prices of individual trading cards have also sky-rocketed, with baseball player Honus Wagner's trading card being sold at $6.6 million in September, 2021. This presents a valuable opportunity for us to tap into the market.

The value of a card depends highly on its condition, yet existing ways to grade the condition of a card are not effective for most transactions. Amateurs are highly inaccurate at grading cards themselves, and expert annotations are too costly and time-intensive for the vast majority of transactions. The result is that millions of trading cards are simply sold as ungraded.

To address this problem, we built a web application that automatically grades trading cards from a picture using machine learning. Sourcing the dataset and working off of Mint Condition[1], a machine learning model that has trained a Convolutional Neural Network (ResNet-18) from scratch, we predict the labels that experts have assigned to training cards. The model was trained on a dataset that has been professionally-graded on Ebay, classifying the cards in 10 categories: Gem-mint, Mint, Near-mint-to-mint, Near-mint, Excellent-to-mint, Excellent, Very-good-to-excellent, Very-good, Good and Fair-to-poor. In addition, we believe a unique feature of our application is that we use the trading card's metadata and classification to retrieve an estimate for market value from EBay.

Our models have proven that training the model from scratch performs better than simply fine-tuning the top layer of a pretrained network, perhaps because low-level visual features such as scratches, printing defects, and rounded corners are very important for grading trading cards.

[1] rthorst. (2020, April 14). rthorst/mint_condition: Automatic Sports Trading Card Grading. GitHub. https://github.com/rthorst/mint_condition

APPROACH

Starting our project, we set several clear objectives to achieve. These can be broken down into the following components of our project:

Backend

The backend goal was straightforward: design a cloud-based architecture leveraging AWS, enabling asynchronous classification of trading cards. However, this can be further broken down into these considerations and requirements:

1. Users will need to authenticate to be able to process a trading card
2. Users will need to store various trading cards and their metadata
3. Users should be able to quickly get a glimpse of card metadata, and condition.
4. Users should be able to monitor the estimated market value of their trading cards over time.
5. Processing should run asynchronously so that the user isn't locked out of the system from some blocking call.
6. Finally, for efficient collaboration, we will consider opportunities to facilitate easy deployments with cloud tools such as CodePipeline.

The architecture for this backend system is illustrated in the Architecture and APIs section.

Frontend

The frontend user interface implements the backend requirements above. The web application allows a user to create an account, upload photo(s) of their trading card(s), classify the condition of their card(s), and view a list of all their uploaded cards and conditions.

The frontend is built using React, a popular frontend framework developed by Facebook and currently used in production by industry leading companies. We chose this frontend framework because it allows reuse of components and enhanced performance via the virtualized DOM feature. React also has good integrations with AWS Amplify, a framework for building and deploying applications.

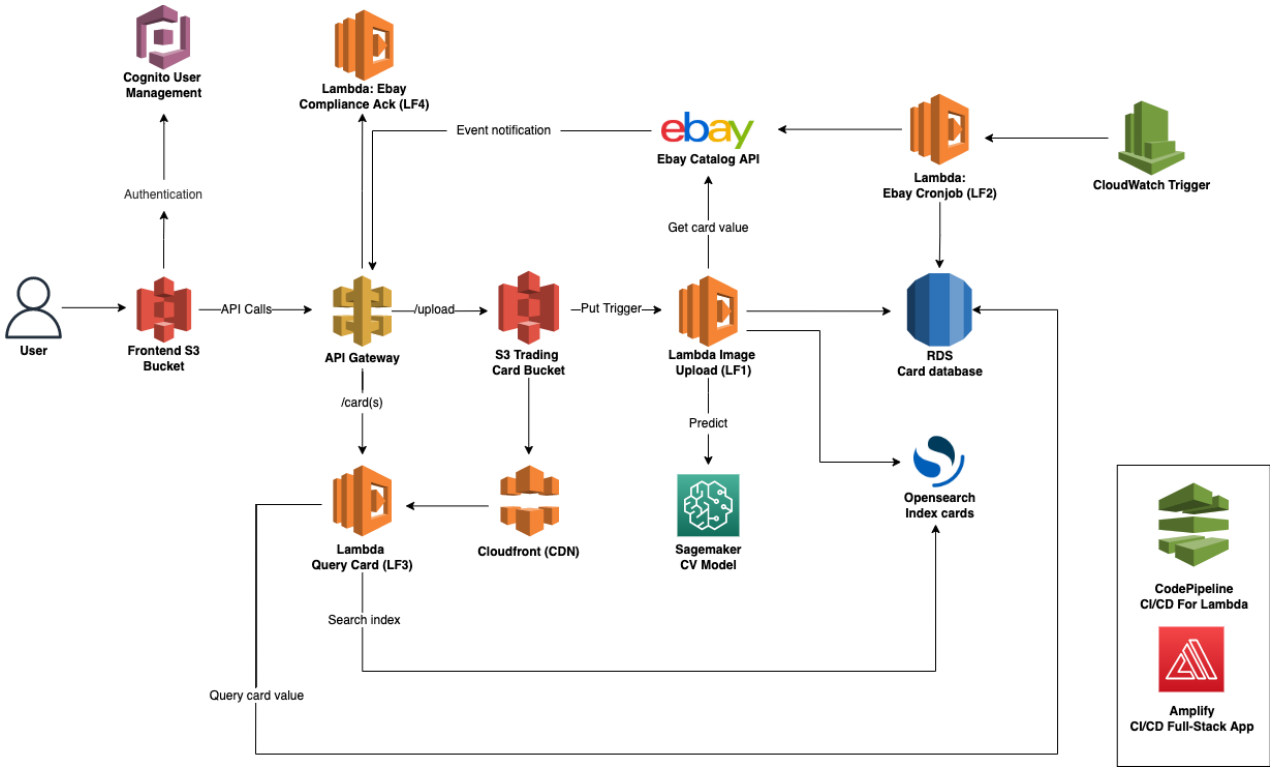
Machine Learning

We planned to use machine learning models to solve the 10-class classification problem. We adopted PyTorch models [4], specifically the Resnet-18 model [5] as the current deployed model. This model achieved comparable performance at the current stage.

We created a training pipeline which is compatible with experiment tracking tools such as Neptune. This pipeline is also capable of developing a model that can determine the condition of a trading card based on a provided image.

We also provided deployment code to achieve end-to-end inference by exploiting AWS Sagemaker. The endpoint contains specific inference functions which are seamlessly integrated with the project system design.

Architecture/APIs



APIs:

- PUT: /upload - upload trading card for classification
- GET: /card/{id} - get detailed information for a specific card
- DELETE: /card/{id} - delete a specific card
- GET: /card/{id}/prices - get all price estimate history for a particular card
- POST: /card/{id}/analyze - re-classify a trading card
- POST: /card - update metadata for specified trading card
- GET: /cards - get all trading cards for user
- GET: /user - get user metadata
- GET: /search - query for trading cards based on labels and condition

The architecture is relatively straightforward. We can break down the architecture based on the APIs. The APIs can be classified into two categories: **queries** and **actions**. Actions include uploading a card, searching for a card, and analyzing a card. The most frequently used action, upload, is handled by S3, passing the request to the first lambda function (LF1) once the file is

saved. **LF1** is charged with coordinating and orchestrating OpenSearch, the DBMS, and Sagemaker for classification. Finally all our card images are available for access via Cloudfront, our CDN.

The query APIs involve retrieving data from the system in some manner. This includes retrieving trading card metadata, user metadata, updating metadata, etc. This is all handled by a single Lambda function **LF3**. In addition, to hit any of these endpoints, users must authenticate with Cognito and pass along any necessary authorization information to API Gateway.

Finally, we have two Lambdas for talking to eBay and retrieving approximate prices for the value of the trading cards in the database. **LF2** can be conceptualized as a cronjob that queries eBay for trading cards to get approximate values. **LF4** is used for authorization with the eBay API.

DESIGN DETAILS & CODE STRUCTURE

Backend

Storage

When a user uploads a card, the card gets uploaded to a private S3 bucket on AWS. The S3 bucket is held private for security reasons. However, the S3 bucket is also connected to Cloudfront (CDN) to effectively cache images in order to speed up their loading and for load balancing.

Authorization and API Gateway

One critical component of our design is allowing users to login and authenticate with the system. We do this to avoid a user being able to access another user's information and trading cards.

Therefore, we use AWS Cognito as our method of authentication. As a bonus, Cognito also provides users the option to use federation identity providers such as Google and Facebook logins.

Once a user either logs in or creates a user with Cognito, they will be provided a unique and temporary identity token which they must use to authorize all their API requests. API Gateway then enforces this requirement and does not allow a user through without the identity token and additional API key.

Database and OpenSearch

Once a user uploads/updates a trading card, our lambda functions coordinate with our database to make any relevant changes. We chose a traditional DBMS running on an EC2 instance. Inside, we ran mariadb, a widely used open source alternative to MySQL. We felt that our data was

highly structured and relational and felt most comfortable working with such systems. In retrospect however, if we truly were worried about immense scalability challenges we should have opted for NoSQL (more on that in the future work section).

Schema

```
DESCRIBE mint_condition.card_conditions;
+------------------------+------------------+------+-----+---------+----------------+
| Field                  | Type             | Null | Key | Default | Extra          |
+------------------------+------------------+------+-----+---------+----------------+
| card_condition_id      | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| card_condition_name    | varchar(100)     | NO   |     |         |                |
| card_condition_descr   | varchar(2000)    | YES  |     | NULL    |                |
| card_condition_rating  | float            | NO   |     | 0       |                |
| card_condition_ml_label | varchar(45)     | YES  |     | NULL    |                |
+------------------------+------------------+------+-----+---------+----------------+

DESCRIBE mint_condition.cards;
+---------------------------+------------------+------+-----+---------+----------------+
| Field                     | Type             | Null | Key | Default | Extra          |
+---------------------------+------------------+------+-----+---------+----------------+
| card_id                   | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| card_label                | varchar(100)     | YES  |     | NULL    |                |
| card_condition_id         | int(10) unsigned | YES  | MUL | NULL    |                |
| user_id                   | varchar(100)     | YES  | MUL | NULL    |                |
| time_created              | int(10) unsigned | YES  |     | NULL    |                |
| card_bucket               | varchar(45)      | YES  |     | NULL    |                |
| card_s3_key               | varchar(145)     | YES  |     | NULL    |                |
| max_notification_threshold | int(10) unsigned | NO   |     | 10      |                |
| min_notification_threshold | int(10) unsigned | NO   |     | 10      |                |
+---------------------------+------------------+------+-----+---------+----------------+

DESCRIBE mint_condition.ebay_price_data;
+--------------------+------------------+------+-----+---------+----------------+
| Field              | Type             | Null | Key | Default | Extra          |
+--------------------+------------------+------+-----+---------+----------------+
| ebay_price_data_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| card_id            | int(10) unsigned | NO   |     | 0       |                |
| timestamp          | datetime         | YES  |     | NULL    |                |
| mean_price         | float            | YES  |     | NULL    |                |
| max_price          | float            | YES  |     | NULL    |                |
| min_price          | float            | YES  |     | NULL    |                |
| count              | int(10) unsigned | YES  |     | NULL    |                |
+--------------------+------------------+------+-----+---------+----------------+
```

OpenSearch is updated anytime the database is. OpenSearch is primarily used for searching the system for a user's cards. We store information such as the condition and card's labels in OpenSearch. Therefore, it is important to maintain consistency between the two metadata stores. As a result, whenever we upload, update, or delete information from the database, we also make sure to do so with OpenSearch. If any of these operations fail, we have implemented failure handling to revert the other associated operation(s) back to the previous state.

After careful analysis, we relaxed our consistency requirements to allow for **loose** or **eventual** consistency between the two stores. Because the database is our ground source of truth, we only pass along information to the user if the data exists in the database. Thus, if a card is removed from the database but not from OpenSearch, from the user's perspective it will still look as if the card has been deleted. This is very effective since guaranteeing strong consistency between the two data stores can be challenging.

Lambda Functions

We implemented four Lambda functions that carry out our processing.

LF1 gets triggered when a trading card is uploaded to the S3 bucket. Details of the card are stored in the relational database. The ML model is invoked to check the condition of the trading card. The label, along with other metadata, returned by the model is stored both in RDS and in OpenSearch.

LF2 is called every 24 hours (at 12 am EST) to check for updated price estimates of the trading cards stored in our database using eBay's Catalog API. Based on user defined labels, the price of such cards is estimated and the database is updated accordingly.

LF3 updates and retrieves information pertaining to the trading cards stored in the database that are associated with a particular user. We retrieve and match user information from Cognito to ensure this. LF3 also re-analyzes the condition of trading cards, if requested by the user and ensures that RDS and OpenSearch are in sync with each other. If not, proper error handling is in place to notify the user and keep the database and OpenSearch consistent.

LF4 is a utility function used to comply with eBay's API specification. In order to move our eBay credentials to production, and access their public data, eBay required us to authorize an API endpoint to receive incoming push alerts. Specifically, two main operations exist in this lambda function. The first function proxies a GET request from eBay to authorize and subscribe the endpoint URL to their service. When successful, eBay periodically sends us POST requests with information regarding account deletion/closure notifications. Essentially, as eBay's API accesses public data (user location, seller information, etc.), when that account is deleted, we would have to delete it in our system, if stored, as well. The lambda function retrieves public keys, and notifies eBay of its receipt.

Frontend

Design

The app was designed in Figma, an online design application that allows users to collaborate in real time, essentially the Google docs of application design. We went for a minimalist design inspired by Material Design, a set of guidelines created by Google, and went with a mint green +

yellow color palette. This was to get as close to a triadic/split complementary color harmony as possible.

Building the Application

React applications are fully written in Javascript which allows us to use a rich ecosystem of tools. Off the bat, we were able to use Yarn to manage dependencies and deploy from any environment with a package.json file. This also allowed us to leverage Create React App, an open source React build tool for single page applications. Having all these tools allowed us to spend most of our time building the application instead of fiddling with dependencies and compatibility.

For implementation, we relied heavily on the many Material Design resources available for React. Many frontend components could be easily imported just like you would import a python library. All that was left for us was some tweaking to fit our design + make the UI flex compatible and we had most of our frontend finished.

Authentication

Since we used AWS Cognito for the backend, we also leveraged the login UI provided by Cognito. When the user clicks the log in/sign up button, we redirect them to the Cognito URL which allows the user to sign up and login. When the user has successfully logged in, cognito returns us to our application and provides an authentication token. With this new token, the user can then access the API and use the app.

As a security measure, the tokens expire after an hour. We handle this on the frontend by sending the user to the login page if we receive an authentication error response from an API request. Since every page needs the user's info, every page makes an API request at least once and guarantees that the user can only see the page if they are logged in.

Backend Integration

Like most webapps, requests to the backend are handled asynchronously so that the user experience does not hang on a failed/pending request. For this, we used Axios as an HTTP client that integrates well with React. We built the UI to have sensible defaults before we receive a response from the API, then populate the UI with React's state management and virtual DOM.

Continuous Deployment

We used AWS Amplify to handle continuous deployment of the app. Amplify reads from Github, builds the app using a config file we provide, and hosts the app on an Amplify server. This is helpful for a few reasons. First, the integrated build environment means we don't need to have our own build server to host a pipeline like Jenkins. All we need to do is provide the build configuration and the rest is handled for us. Second, the application is hosted for free on AWS

servers and page routing is abstracted away for us. Our app has several pages which means we would not be able to statically host it in S3. Instead of having to deploy our app to a server however, Amplify's hosting environment makes the process very easy and even provides a domain for users to visit our site. In production, we would update this domain with a purchased domain but for our purposes, this was extremely useful for rapid development and deployment.

Machine Learning

Dataset and Evaluation Settings

The TradingCards dataset contains a total of 87,354 professionally-graded cards, and 10 classes from Ebay auctions. To train our TradingCardsNet and do the evaluation, we split the training phase and testing phase into 9:1, and use accuracy, loss, F1-score, and Pearson score for the overall performance.

Augmentation and Preprocessing

Rotation: Image rotation is one of the most commonly used augmentation techniques. It can help our model become robust to the changes in the orientation of the cards. For our use case, even if we rotate the image, the classification of the image should remain the same. In our implementation, every image is randomly rotated and used as a data point.

Normalization: Subtracting the dataset mean from the dataset serves to "center" the data. Additionally, we would ideally like to divide by the standard deviation of that feature or pixel as well. The reason we do both of those things is because in the process of training our network, we're going to be multiplying (weights) and adding to (biases) these initial inputs in order to cause activations that we then backpropagate with the gradients to train the model. We'd like in this process for each feature to have a similar range so that our gradients don't go out of control (and that we only need one global learning rate multiplier).

Mixup: Mixup is a domain-agnostic data augmentation technique proposed in "Mixup: Beyond Empirical Risk Minimization" by Zhang et al. It's implemented with the following formulas:

$$\tilde{x} = \lambda x_i + (1-\lambda)x_j, \qquad \text{where } x_i, x_j \text{ are raw input vectors}$$
$$\tilde{y} = \lambda y_i + (1-\lambda)y_j, \qquad \text{where } y_i, y_j \text{ are one-hot label encodings}$$

(Note that the lambda values are values with the [0, 1] range and are sampled from the Beta distribution.) The technique is quite systematically named - we are literally mixing up the features and their corresponding labels.

Neural networks are prone to memorizing corrupt labels. Mixup relaxes this by combining different features with one another (same happens for the labels too) so that a network does not get overconfident about the relationship between the features and their labels. Mixup is

specifically useful when we are not sure about selecting a set of augmentation transforms for a given dataset.

Evaluation Metrics

We adopt accuracy as our main evaluation metric as we treat the problem as a 10-class classification problem.

Pytorch Lightning & Neptune

We chose to implement our training with pytorch-lightning which is an extended framework integrated with pytorch with quick Neptune support. Our training code can be found under LINK. The training pipeline integrates multiple models including ResNet18, VGG16, SqueezeNet etc.

The following is the design of the training model class:

```
class TradingCardsNet(LightningModule):
   def __init__(self, **kwargs):
       super().__init__()
       self.model = kwargs.get('model', 'resnet18')
       fine_tuning = kwargs.get('fine_tuning', True)
       pretrained = kwargs.get('pretrained', True)
       self.lr = kwargs.get('lr', 1e-3)
       ...
   def forward(self, x):...
   def configure_optimizers(self):...
   def training_step(self, batch, batch_idx):...
   def training_epoch_end(self, outputs):...
   def validation_step(self, batch, batch_idx):...
   def get_progress_bar_dict(self):...
```

Sagemaker

We deploy the trained pytorch lightning model in Sagemaker. Note that Pytorch lightning is not compatible with Sagemaker, we use the following snippet to convert lightning model to torch sequential model:

```
model = TradingCardsNet.load_from_checkpoint(checkpoint_path=chkpt_path)
seq = nn.Sequential(model.feature_extractor, model.classifier)
torch.save(seq, "best_model.p")
```

The file "best_model.py" contains the deployable model for Sagemaker using PytorchModel[2] wrapper as the following:

```
PyTorchModel(
    model_data="[s3 model tar.gz file]",
    role=role,
    entry_point='[inference script]',
    framework_version='1.5.0',
    py_version='py3',
    model_server_workers=8
).deploy(
    initial_instance_count=1,
    instance_type='[instance type]'
)
```

The inference script contains four functions: input_fn, model_fn, predict_fn and output_fn to determine the endpoint behavior. Each time an endpoint is called, these four functions are called sequentially. We follow the same directory structures suggested here[3] to wrap up the resource package. We have also adopted S3 support whenever an user updates an image data. Instead of passing the image data directly to our function, we only ask clients to send a request with S3 buckets and keys. In this way, we can avoid duplicated data transmission and reduce the prediction service latency.

```
def model_fn(model_dir):
    """Creates pytorch model specified via model_dir.

    Args:
        model_dir (str): path to model root dir.
    """

def predict_fn(input_object, model):
    """ Uses Image library to load images and transform them into float
numbers for model prediction.

    Args:
```

[2] PyTorch — sagemaker 2.72.1 documentation. (2021). Readthedocs.io. https://sagemaker.readthedocs.io/en/stable/frameworks/pytorch/sagemaker.pytorch.html

[3] Use PyTorch with the SageMaker Python SDK — sagemaker 2.72.1 documentation. (2021). Readthedocs.io. https://sagemaker.readthedocs.io/en/stable/frameworks/pytorch/using_pytorch.html#create-the-directory-structure-for-your-model-files

```
        input_object (Image): image loaded using PIL.Image
        model (Object): unpickled model provided using model_fn
    """

def input_fn(request_body, request_content_type):
    """ Loads input from request content and returns image specified in
request_body as PIL.Image.

    Args:
        request_body (bytes): serialized dictionary with two required keys:
"bucket" and "key" which specify an image to predict.
        request_content_type (str): type of content
    """

def output_fn(prediction, content_type):
    """Outputs the result of prediction as a dictionary response.

    Args:
        prediction (str): Label predicted by predict_fn
        content_type (str): type of content
    """
```

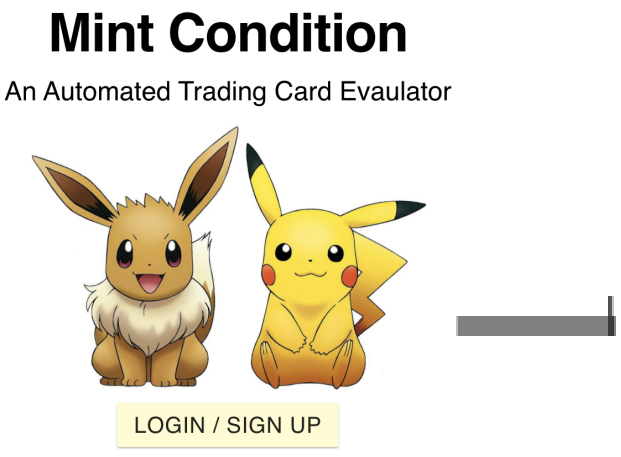
RESULTS

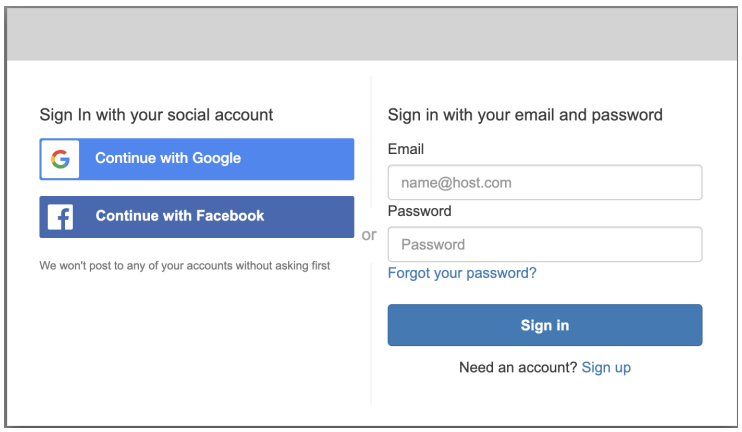
Web Application

These are the key features we have implemented on the frontend:

1. Login authentication through Cognito

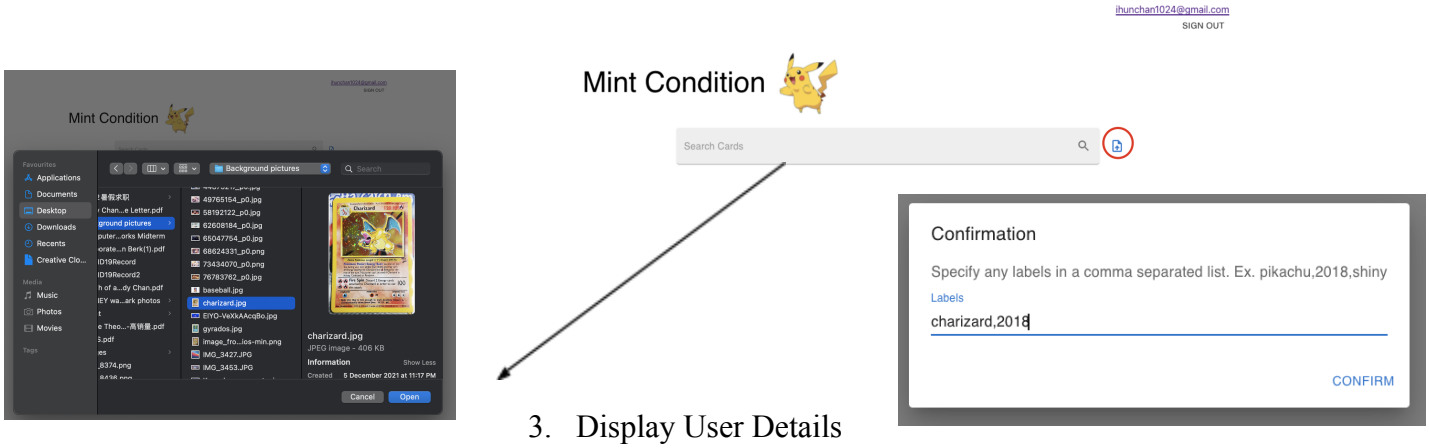
Clicking on the LOGIN / SIGN UP button will lead the user to the cognito sign in page.





2. Upload trading card & corresponding label

Upload a card by clicking on the upload button beside the search bar, and specify labels after selecting the card to upload.



3. Display User Details

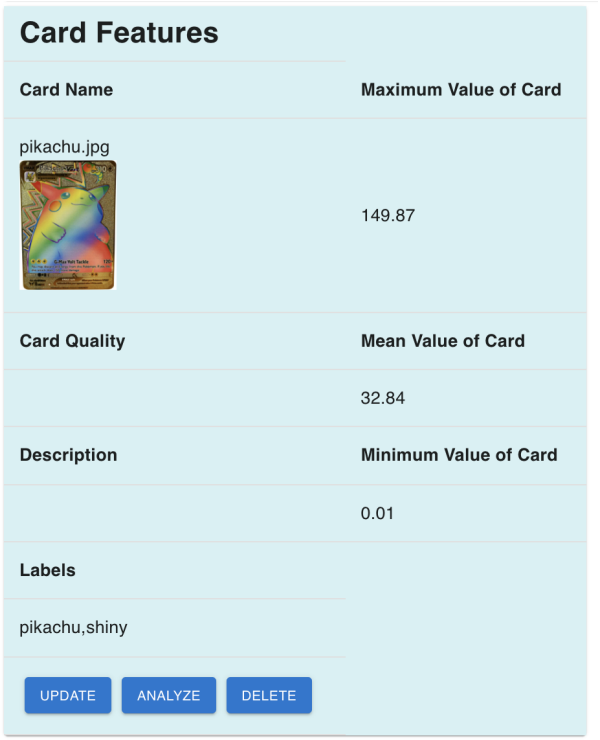
User details are displayed by clicking on username on the top right corner of the page.

User Profile

Name	Number of User Cards
ihunchan1024@gmail.com	3
User Purpose	**Estimated Value of Cards**
Trade	135.81

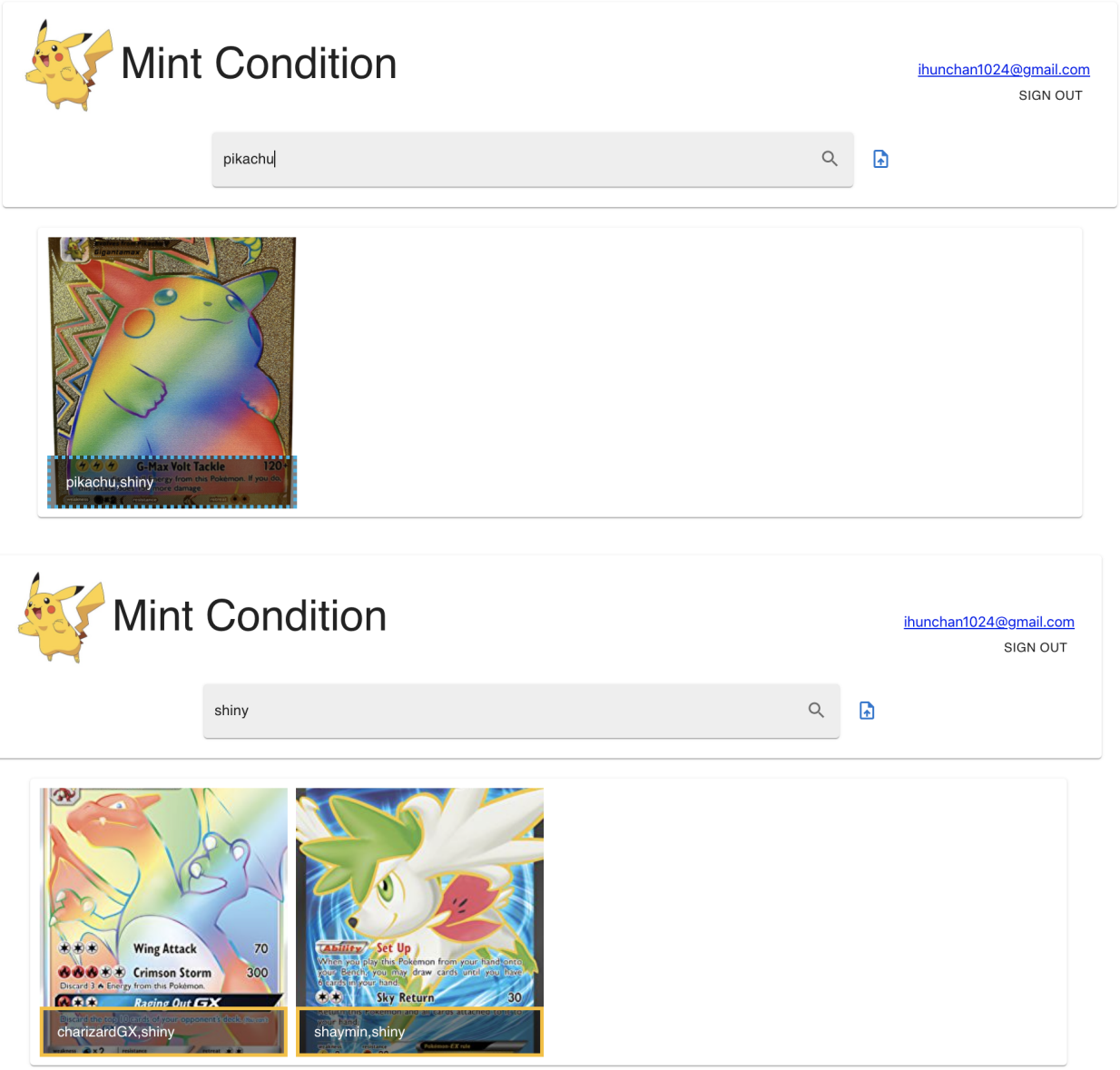
4. Display Card Details

Card details are displayed by clicking on each individual card.



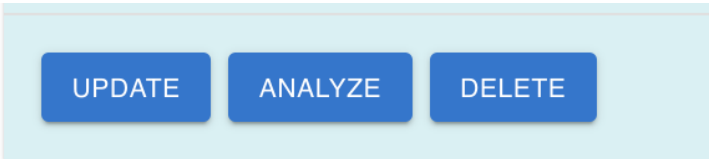
5. Search & display card based on labels and condition

Users can search for cards based on their labels or condition. To specify a condition, the user just needs to include a comma. For example, “Pikachu, Good” will search for all cards labeled Pikachu in a “Good” condition.



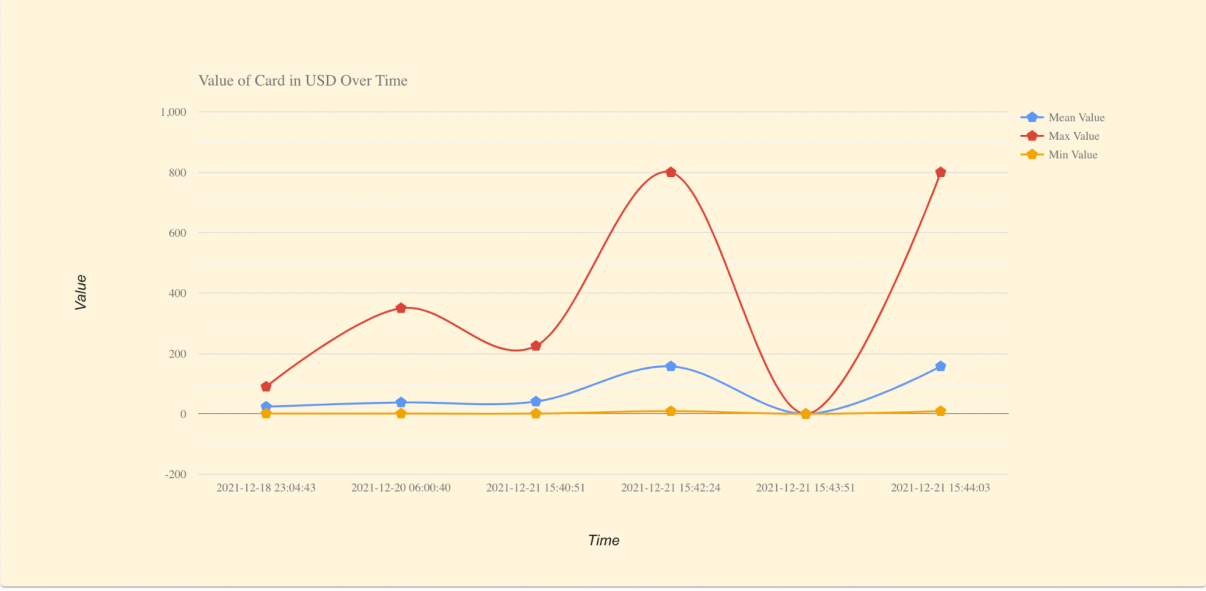
6. Update, Delete, and Analyze cards

Buttons are provided for updating a card's labels, reanalyzing a card if the user is unhappy with the classification, and deleting a card from the system.



7. Card Price History

We also graph the price history of a card over time. Anytime a label changes, or once every 24 hours, new price history is added and visualized in the frontend.



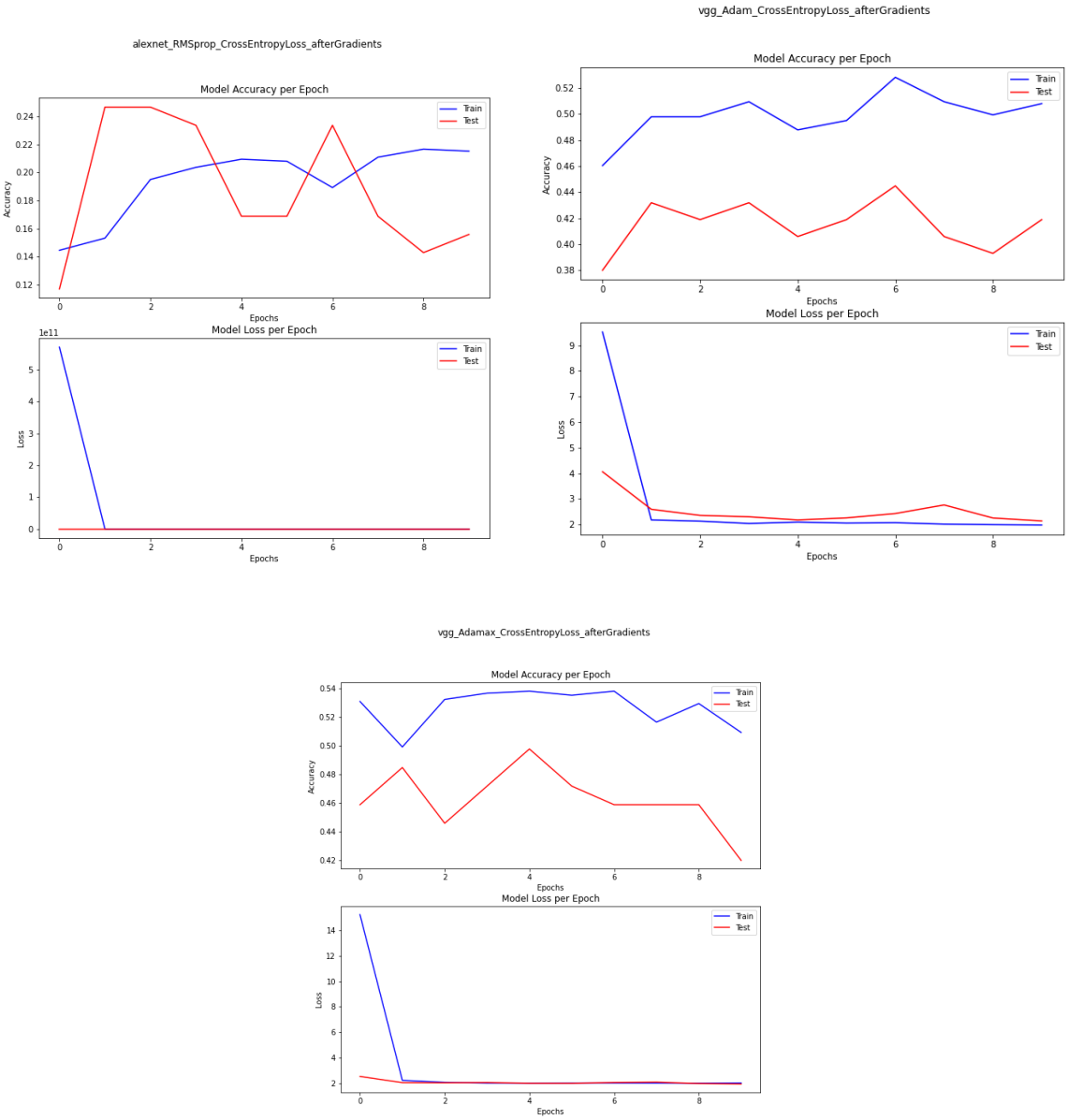
Machine Learning

We separated our Machine Learning experiments into several phases in order to methodically arrive at the most optimal model combination for the task of classifying trading cards according to their condition. We first tested various optimization techniques while training the models themselves in order to arrive at a training pipeline that was not only efficient timewise (allowing us to train and test more models in the same period of time) but also that allowed for improved classification accuracy before even considering the model improvements. After these optimizations were honed in on, the next phase was selecting the appropriate combination of model, optimization function and loss function so that we could arrive at a model whose accuracy was noticeably higher without overfitting to training data. In order to perform these comparisons across models and different optimizations, it was key to our training pipeline that we constantly saved training progress of various models in the form of performance metrics, model checkpoints and epoch history - each in their own subdirectory - so as to avoid confusion at a later stage.

Finally, we used Neptune to keep track of various model hyperparameters such as batch size and learning rate so that once we arrived at the final model, we could optimize on the minutia of the model to squeeze the most out of its performance. We chose this approach of model selection to avoid unnecessary excessive computation or settling at a local minima by considering too many variables at once when comparing models.

Fine tuning the last layer (blue) vs fine tuning the whole model (purple)

One of the optimizations we tried with our models was to first only allow the neurons in the top layer of the model to return a gradient. The hope was that this would allow the first layer to learn basic features of the card classification while later layers the gradients of which were enabled in a subsequent run could adapt to more abstract features. The thought was that this might avoid overfitting to a local optima and as we can see from just one of the examples of model combinations, (VGG with an AdaMax Optimization function and CrossEntropy Loss), the accuracy and loss both perform better (within reason given this model combination) after this optimization than before. Both the accuracy and the loss on the training data are more stable and reach better values after enabling the rest of the gradients while we notice that the test data metrics are also less volatile.



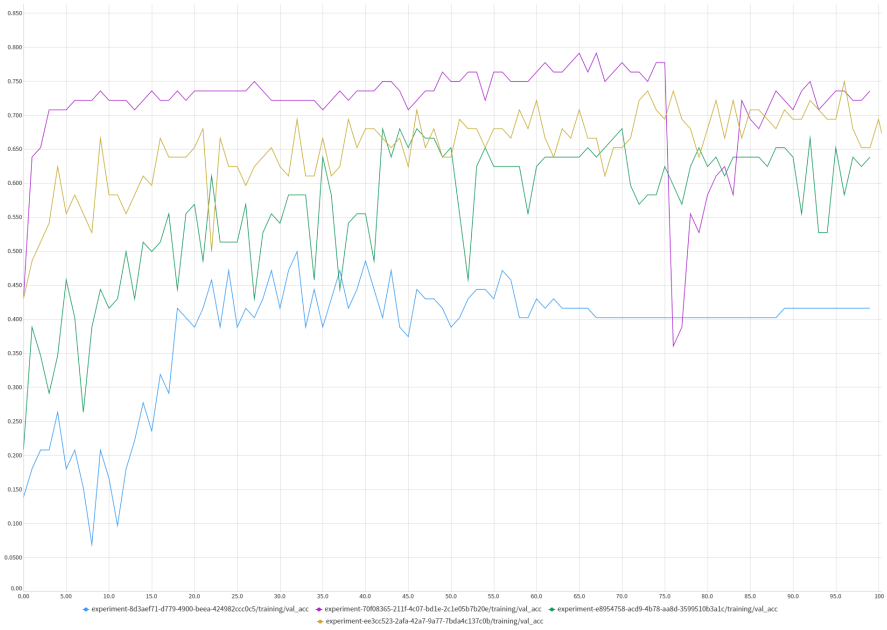
Model performance across different optimizers

From the above diagrams and more so, our observation of model classification in practice, we noticed that certain models were performing the most consistently - largely due to the nature of their original purposes. In addition, it was noted that CrossEntropy Loss performed most consistently and that the AdaMax and Adam optimization functions were both the best optimization functions with the AdaMax optimization function reaching slightly higher peaks on certain models.

Finally after the above phases, we selected the VGG 19 with batchnorm model, using CrossEntropy Loss, Adam optimization function and went about tuning hyperparameters to

achieve the following results:

batch size		4	8	16	32	64
AlexNet	lr=0.0001	0.44444	0.54166	0.44444	0.33333	0.47222
	lr=0.001	0.14283	0.13888	0.51388	0.26388	**0.55555**
VGG19 bn	lr=0.0001	**0.80555**	0.71234	0.70833	0.69444	0.72222
	lr=0.001	0.45833	0.625	0.5	0.38888	0.5
Resnet18	lr=0.0001	**0.764**	0.653	0.555	0.514	0.417
	lr=0.001	0.486	0.611	0.611	0.528	0.555
SqueezeNet	lr=0.0001	**0.639**	0.542	0.514	0.555	0.431
	lr=0.001	0.403	0.347	0.388	0.319	0.264

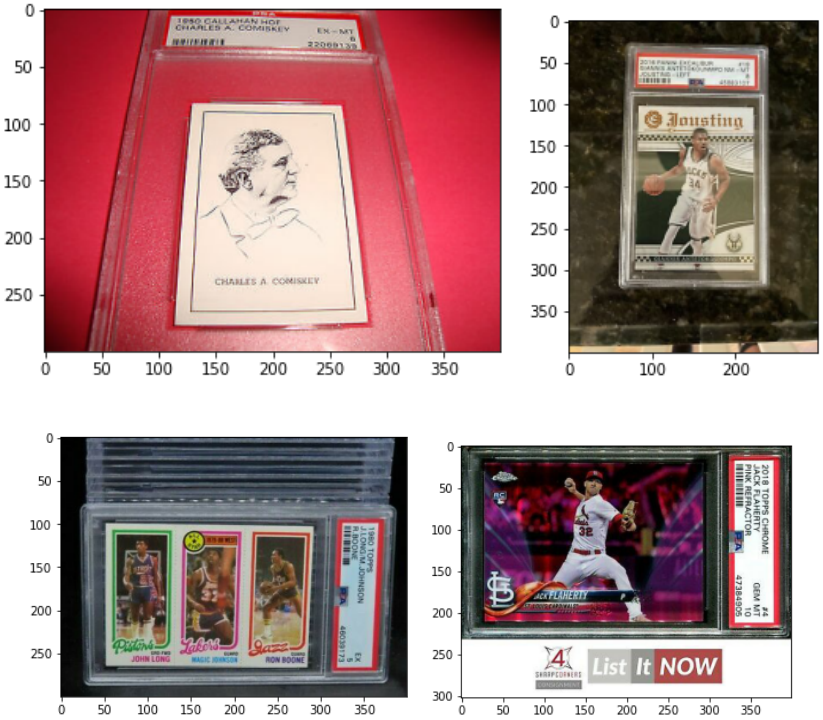


The validation accuracy of 4 different architectures. The purple curve is VGG19, yellow curve is Resnet18, green curve is SqueezeNet, blue curve is AlexNet.

FOLLOW UP WORK FOR MACHINE LEARNING

Observation

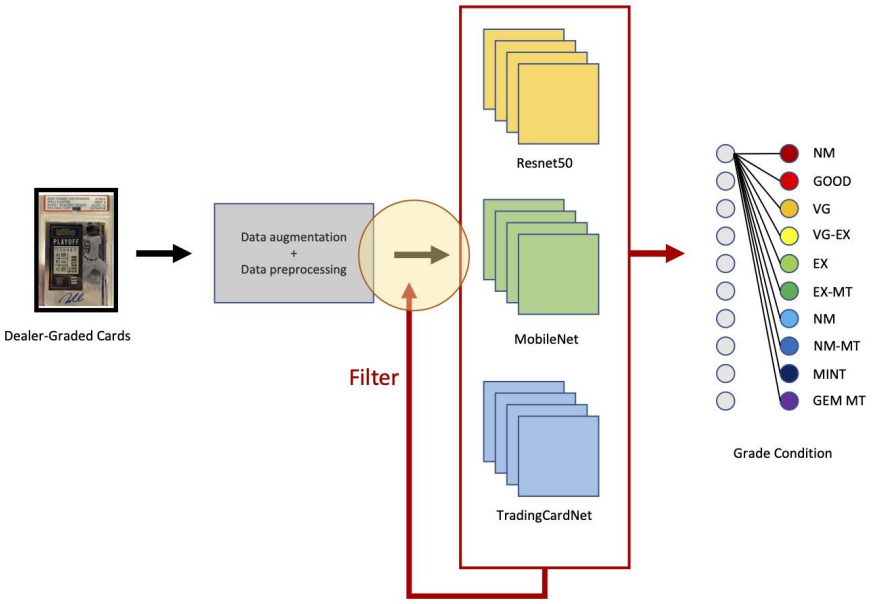
Because the accuracy of the TradingCardNet only achieves 80.5%, we tried to delve into the failure cases of the experiment and these failure cases are shown below:



We can find that all failure cases have noisy backgrounds. In these cases, they not only contain the trading card itself but also include lots of background information to distract the classification models.

New training pipeline

To deal with this kind of problem, we set up a new training process to remove these noisy backgrounds and then use the new dataset to train a new model, the pipeline is as follows:



In this new process, we will first train a TradingCardNet and make sure it would not overfit with the dataset. Then, we will use this TradingCardNet as a filter, removing all images in the training dataset which the filter predicts to the wrong class. Finally, we will use this new dataset to train a final new TradingCardNet as our trading card classification model.

After we use this new approach to train our TradingCardNet with VGG 19 structure, 4 batch size, and 0.0001 learning rate, the accuracy is **75%**. Which means this approach cannot enhance the performance of our trading card task. The details of the training process are shown as below:

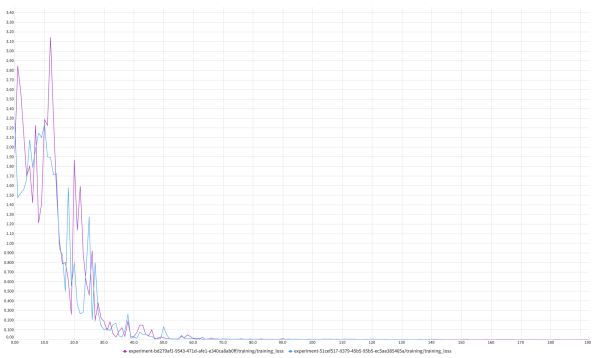


Figure a. The training loss of two approaches

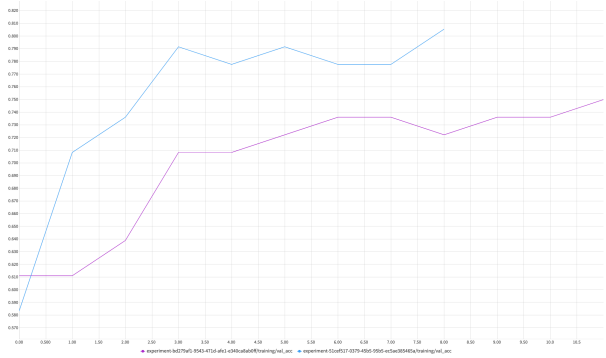


Figure b. The validation accuracy of two approaches

For these two figures, the blue curve represents the original TradingCardNet with early stopping, and the purple curve represents the follow up new TradingCardNet with early stopping. We can find that the new approach can not get better results. After studying this filter in depth, we found

that this filter cannot filter out all the noisy images. We summarized the reasons why the accuracy cannot be improved:

1. The filter cannot filter all the noisy images, for example, the noisy trading card images including the noisy background, the trading card will reflect light, there are multiple cards in the same picture and only one label is provided
2. The filter is trained from the original noisy trading card dataset, and it overfits during the training process. This filter cannot filter all the noisy images from the trading card training dataset.
3. We use the same testing dataset to make a fair comparison between the original TradingCardNet and new approach. However, the new TradingCardNet has never seen noisy data before, and it also predicts wrong results when dealing with noisy data in the testing dataset.

Finally, we also experimented with the filter that removed all noisy data in the training set and testing set to prevent the third problem discussed earlier. The result shows that the new TradingCardNet can achieve **89.47%** accuracy in the new testing dataset.

FUTURE WORK

Backend

In the future we would like to make the backend a bit more robust in error handling scenarios. Although we did implement some reverting in case of errors, there are a number of ways the various components could fail and the backend could keep on operating without notifying the client of an error. In particular, the Sagemaker logic has not been hardened to various failure scenarios.

We would like to add a manual scan of the photo (using a new endpoint) in case the model produced some kind of failure. That way, the client could decide how to handle the situation instead of the backend.

For the DBMS we would like to enhance the design of the architecture by implementing a database cluster design. One of the more common ones is Galera. However, we also know that NoSQL is the path to true scalability and High Availability so we may decide to re-architect the entire database backend with that in mind. Potential replacements would be DynamoDb or MongoDB.

To provide an even better experience for the user, we could additionally add an alerts feature to the site, where users specify a threshold on card values, and be notified when that threshold is met, during the daily cron jobs. For instance, a user could be in search of selling their Pikachu card on eBay. Having received a grade prediction with our model, and an estimated price range

for that specific card, that user could set a notification for when the price of the card increases by 20%. That could be the user's moment for them to list the card on eBay as well.

Frontend

Any changes in the backend would result in corresponding changes in the frontend.

Besides this, there are a few things we would like to polish and extend to the frontend:

1. We want to allow users to compare different cards by including a compare option (compare & contrast value of two different cards, possibly in the user page, or similar cards we can scrap from online), this will give the user a better grasp of the value of the card in selection.
2. Modify platform to support chat & user interactions. (users can view each other's cards and card values) -> we can further support user card trades through the platform.
3. Support cross-platform user interactions - can automatically detect users who are friends with the current user on Facebook etc.

Machine Learning

Our model was able to achieve an accuracy of 80% after all efforts were made. In order to achieve a higher accuracy the following future work can be completed:

1. New training set with higher resolution photos
    a. Photos should be consistent and not of various zoom levels
    b. Photos needs to be less noisy - preferably showing only a single object
    c. Increase the number of photos on training by building a realtime streamer
2. To further increase the model performance in the future, one possible modification is to remap the problem to other domains apart from classification problem, e.g., logistic regression. Other high level feature embedding options are also available.
3. Adopt better instances for setting up sagemaker GPU-training.

SUMMARY

We have successfully deployed an easy-to-manage dynamic web app on AWS amplify. The web app supports the ability to upload, update, and analyse various types of trading cards, and provides users with thorough details about the uploaded cards. This is all facilitated with a robust backend, where we provide relevant user and card information via API requests and handle incoming data for classification asynchronously. In addition, our cron job for retrieving estimated card values makes our application especially unique and powerful. The backend is facilitated by

a machine learning model deployed to AWS Sagemaker that is able to classify card conditions with up to 80% accuracy.

SOURCES

1. Gastelum, A. (2021, August 16). Honus Wagner Card Sells for $6.606 Million, Becomes Most Expensive Trading Card Ever. Sports Illustrated; Sports Illustrated. https://www.si.com/mlb/2021/08/16/t206-honus-wagner-baseball-card-breaks-record-sale
2. Mueller, R. (2021, August 15). eBay: 2021 Trading Card Sales Have Already Passed All of 2020. Sports Collectors Daily; Sports Collectors Daily. https://www.sportscollectorsdaily.com/ebay-trading-card-sales-2021-half/
3. Young, J. (2021, September 30). Explaining the whopping $10 billion valuation for Fanatics' nascent trading card business. CNBC; CNBC. https://www.cnbc.com/2021/09/30/explaining-fanatics-whopping-10-billion-valuation.html
4. https://pytorch.org/vision/stable/models.html
5. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. CoRR, abs/1512.03385. Opgehaal van http://arxiv.org/abs/1512.03385